SANDIA REPORT

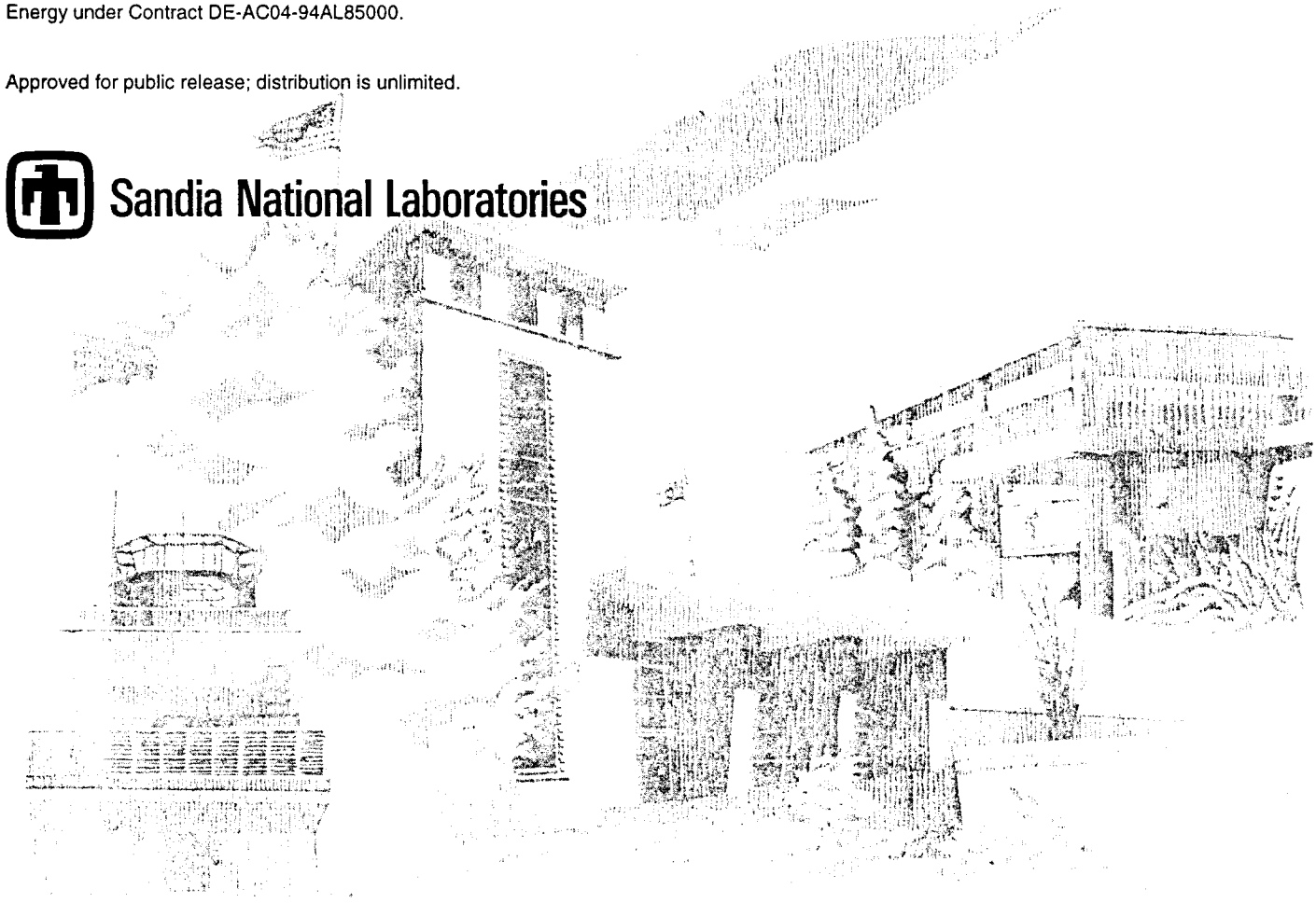
SAND98–8206 • UC–405
Unlimited Release
Printed November 1997

Jess, The Java Expert System Shell

Ernest J. Friedman-Hill

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

Distribution
Category UC-405

SAND98-8206
Unlimited Release
Printed November 1997

Jess, The Java Expert System Shell

http://herzberg.ca.sandia.gov/jess

Ernest J. Friedman-Hill
Scientific Computing Department
Sandia National Laboratories
Livermore, CA

Version 3.2 (October 8, 1997)

ABSTRACT

This report describes Jess, a clone of the popular CLIPS expert system shell written entirely in Java. Jess supports the development of rule-based expert systems which can be tightly coupled to code written in the powerful, portable Java language. The syntax of the Jess language is discussed, and a comprehensive list of supported functions is presented. A guide to extending Jess by writing Java code is also included.

1 Introduction

Jess is a clone of the popular expert system shell CLIPS, rewritten entirely in Java. With Jess, you can conveniently give your Java applets and applications the ability to 'reason'. In describing Jess, I am going to describe much of CLIPS itself, but the reader may want to obtain a copy of the CLIPS manuals available. See the WWW site http://www.ghg.net/clips/CLIPS.html for more information about CLIPS. Note that Jess does not duplicate all of CLIPS, but only the essential core of it.

Jess is compatible with all versions of Java starting with version 1.0.2. It is (in particular) Java 1.1 compatible, although while compiling you will see warnings about deprecated methods. Such is the price of compatibility!

Jess is a work in progress - more features are always being added. The order will be determined in part by what folks seem to want most, what I need Jess to do, and how much time I have to spend on it.

This is the 3.2 final release. Nevertheless, there may well still be bugs. Please report any that you find to me at ejfried@ca.sandia.gov so I can fix them for a later release.

Jess is copyrighted software - see the file LICENSE for details.

1.1 Getting Started With Jess

If you download Jess for UNIX, you can extract the files using tar and uncompress:

```
uncompress Jess-3.2.tar.Z
tar xf Jess-3.2.tar
```

If you downloaded Jess for Windows, you get a .zip file which should be unzipped using a Win32-aware unzip program like WinZip. Don't use PKUNZIP - it cannot handle long file names. WinZip is nice.

When Jess is unpacked, you should have a directory named 'Jess32'. Inside this directory should be the following files:

README.html	This file
TextAreaOutputStream.java Jess.java LostDisplay.java NullDisplay.java	Java source files. Jess.java implements both the applet interface and the command-line interface. NullDisplay is used by the command-line version; the other two are used in the applet's GUI.
jess/	A directory containing the 'jess' package. There are many source files in here that implement Jess's inference engine.
examples/	A directory of tiny example CLIPS files.
index.html	A web page containing the Jess example applet; it may need to be edited!
Makefile	A simple makefile for Jess.

Jess comes as a set of Java source files. You'll need to compile them first: the commands

```
javac *.java jess/*.java (UNIX)
```

or

```
javac *.java jess\*.java (Win32)
```

would work just fine, given that you have a Java compiler like Sun's JDK. If you have problems, be sure that the directory in which the file Jess.java exists is on your CLASSPATH; this may mean including '.' (dot). You can use either a Java 1.0.2 or a Java 1.1

compiler to compile Jess; the resulting code runs on either 1.0 or 1.1 VMs. Note that if you use a 1.1 compiler, you will see some warning about 'deprecated methods' - it is safe to ignore these warnings. I could make them go away, but then Jess would not be 1.0.2 compatible!

There is one optional source file in the subdirectory Jess32/jess/view/. This file defines the optional debugging command 'view'. It can be compiled only with Java 1.1 or later.

There are several example programs for you to try. They are called `fullmab.clp`, `zebra.clp`, and `wordgame.clp`. `fullmab.clp` is the Monkey and Bananas problem featured at the Jess web site. To run it yourself from the command line, just type

```
java Jess examples/fullmab.clp (or examples\fullmab.clp on Win32)
```

and the problem should run, producing a few screensfull of output. Any file of CLIPS code (given that it contains only CLIPS constructs and functions implemented by Jess, as described in this document) can be run this way. Note that giving Jess a file name on the command line is like using the 'batch' command in CLIPS; therefore, you need to make sure that the file ends with

```
(reset)
(run)
```

or nothing will happen. `zebra.clp` and `wordgame.clp` are two other classic CLIPS examples, slightly modified to run under Jess. Both of these examples were selected to show how Jess deals with tough situations. These examples both generate huge numbers of partial pattern matches, so they are slow and use up a lot of memory. They may each take tens of seconds to run, depending on your computer, but they will run.

Jess now has an interactive command-line interface, which has been improved for Jess 3.1. Just type `java Jess` to get a 'Jess>' prompt. In support of this, there is now an (exit) command. To execute a file of CLIPS code from the command prompt, use the 'batch' command:

```
Jess> (batch myfile.clp)
  (lots of output)
```

Jess also now sports a 'system' command, which means, for example, that you can invoke an editor from the Jess command line to edit a file of Jess code before reading it in with 'batch'. 'system' will also help to allow non-Java programmers to integrate Jess with other applications. Given that you have an editor named 'notepad' on your system, try

```
Jess> (system notepad README)
  TRUE
```

The class 'Jess', which contains the main routine that allows you to execute CLIPS code from the command line, also implements an Applet interface, so that it will run in a Web browser. The Applet interface is specialized to run only the 'mab.clp' Monkey and Banana example. To create your own graphical applets using the Jess classes, read on, and check out the Sections about calling Jess from Java and vice-versa. You can modify the Jess class, or you can write your own from scratch (which is probably a better idea.)

2 Major Jess Features

Jess implements the following constructs from CLIPS: defrules, deffunctions, defglobals, deffacts, and deftemplates. Jess has none of the object-oriented CLIPS extensions: defclass, defgeneric, etc. are *not* included. Jess does not implement modules, either. However, since Jess is object-oriented, you can instantiate multiple Jess systems and get them to communicate via the external function interface (see Section 8, Extending Jess with Java).

Jess supports the following basic data types: SYMBOL, STRING, INTEGER, FLOAT, FACT_ID, and EXTERNAL_ADDRESS. In addition, values of the following types are used internally by Jess: VARIABLE, FUNCALL, ORDERED_FACT, UNORDERED_FACT, LIST, DESCRIPTOR, INTARRAY, and NONE. Note that all Jess numbers obtained via scanning textual input become FLOATs. INTEGERs may be returned by functions, however.

Because of the way Jess compiles rules, it works much better if you define your rules first before loading in facts. Deffacts don't count as loading in facts, but the (reset) and (assert) commands do. Put your rules first in your Jess input files for the most efficient operation of Jess.

Jess implements only a small subset of CLIPS intrinsic functions. These are functions which are essentially built into Jess and cannot be removed. All of these have been designed to function as much like their CLIPS counterparts as possible. The currently supported intrinsic functions are

```
*, +, -, /, <, <=, <>, =, > , >=, and, assert, assert-string, bind,
clear, eq, exit, facts, gensym*, halt, if, jess-version-number,
jess-version-string, load-facts, mod, modify, neq, not, or, printout,
read, readline, reset, retract, return, rules, run, save-facts,
sym-cat, undefrule, unwatch, watch, while .
```

On the other hand, I'm supplying implementations for many more CLIPS functions as 'Userfunctions' - external functions written in Java that you can plug into Jess. See the files `jess/StringFunctions.java` (string handling functions: str-cat, str-compare, etc), `jess/MultiFunctions.java` (multifield functions: create$, nth$), `jess/PredFunctions.java` (predicates: oddp, stringp, etc), `jess/MiscFunctions.java` (batch, system), and `jess/MathFunctions.java` (abs, sqrt) for more information. All of the included Userfunctions are installed into the command-line version of Jess by default; you can pick and choose in your own applications. In applets, in particular, you may want to include only the Userfunctions you need, to keep the size of the applet down. (see Section 8, Extending Jess with Java, for information about doing this.)

Here is the complete list of Userfunctions shipped with Jess 3.2:

```
**, abs, batch, complement$, create$, delete$, div, e, evenp, exp,
first$, float, floatp, implode$, insert$, integer, integerp,
intersection$, length$, lexemep, load-function, load-package, log,
log10, lowcase, max, member$, min, multifieldp, nth$, numberp, oddp,
pi, random, replace$, rest$, round, setgen, sqrt, str-cat,
str-compare, str-index, str-length, stringp, sub-string, subseq$,
subsetp, sym-cat, symbolp, system, time, union$, upcase .
```

All these functions are described in detail later in this document.

One more note - in the interest of size and speed, Jess assumes that your input code is largely correct CLIPS code. As a result, if your CLIPS code is syntactically invalid, Jess's error messages may be less than helpful. It certainly would help you develop Jess code if you had a copy of CLIPS to test on.

3 The Jess Language

Jess is effectively an interpreter for a rule language borrowed from CLIPS. I will briefly describe this language here; more information can be gotten from the CLIPS manuals themselves.

I'm using an extremely informal notation here to describe syntax. Basically strings in <angle-brackets> are some kind of data that must be supplied; things in [square brackets] are optional, and ellipses (...) are used to indicate one or more of the preceding. In general, input to Jess is free-format; newlines are generally not significant and are treated as whitespace.

In the example dialogs, You type what appears after the *Jess>* prompt. The system responds with the text in **bold.**

3.1 Atoms

An 'atom', or symbol, is a common concept in the Jess language. Atoms are very much like identifiers in other languages. A Jess atom can contain letters, numbers, and the following punctuation: $*=+/<>_?#. . An atom may not begin with a number; it may begin with some punctuation marks (some have special meanings as operators when they appear at the start

of an atom.) The best atoms consist of letters, numbers, underscores, and dashes; dashes are traditional word separators. The following are all valid atoms:

```
foo   first-value   contestant#1   _abc
```

3.2 Numbers

Jess parses numbers using the Java StreamTokenizer class. Therefore, it accepts only simple floating point and integer numbers; it does not accept scientific or engineering notation. The following are all valid numbers:

```
3   4.   5.643
```

3.3 Strings

Character strings in Jess are denoted using "double quotes." Backslashes can be used to escape embedded quote symbols. The following are all valid strings:

```
"foo"      "Hello, World"      "\"Nonsense\," he said firmly."
```

3.4 Lists

The fundamental unit of syntax in Jess is the list. A list always consists of an enclosing set of parentheses and zero or more atoms, numbers, strings, or other lists. The following are valid lists:

```
(+ 3 2)   (a b c)   ("Hello, World") ()   (deftemplate foo (slot bar))
```

The first element of a list (the 'car' of the list in LISP parlance) is often called the list's 'head' in Jess.

3.5 Comments

Programmer's comments in Jess begin with a semicolon (;) and extend to the end of the line of text. Comments cannot appear inside of constructs (see Section 3.8, Constructs). Here is an example of a comment

```
; This is a list
(a b c)
```

3.6 Functions

Jess contains a large number of built-in functions that you may call; more functions are provided as extensions. You can write your own functions in the Jess language (see Section 3.9, Deffunctions) or in Java (see Section 8, Extending Jess with Java.)

Function calls in Jess use a prefix notation. A list whose head is an atom that is the name of an existing function can be evaluated as an expression. For example, an expression that uses the "+" function to add the numbers 2 and 3 would be written (+ 2 3). When evaluated, the value of this expression is the number 5 (not a list containing the single element 5!) In general, expressions are recognized as such and evaluated in context when appropriate. You can type expressions at the Jess> prompt; Jess evaluates the expression and prints the result.

```
Jess> (+ 2 3)
  5.0
Jess> (+ (+ 2 3) (* 3 3))
  14.0
```

Note that all arithmetic results are returned as floating-point numbers; all arithmetic is done as floating-point by Jess. A comprehensive list of functions implemented in Jess, with descriptions, is given in Section 4.

3.7 Variables

Programming variables in Jess are atoms that begin with the question mark (?) character. The question mark is part of the variable's name. A normal variable can refer to a single atom, number, or string; a variable whose first character is instead a "$" (for example, $?X) is a 'multivariable', which can refer to a list of items. You assign to a variable using the 'bind' function:

```
(bind ?x "The value")
```

Variables need not (and cannot) be declared before their first use.

3.8 Constructs

Besides expressions, the Jess language includes another kind of special list called a 'construct.' A construct is a list that defines something to the Jess system itself. For example, the deffunction construct is used to define functions (see Section 3.9, Deffunctions.) A construct evaluates to TRUE if it was accepted, or FALSE if it was not.

3.9 Deffunctions

The deffunction construct is used to define functions that you can then call from Jess. A deffunction construct looks like this:

```
(deffunction <function-name> ([<parameter1> [<parameter2> [...]]])
  [<doc-comment>]
  [<expr1> [<expr2> [...]]]
  [<return-specifier>])
```

The <function-name> must be an atom. Each parameter must be a variable name (all functions use pass-by-value semantics). The optional <doc-comment> is a double-quoted string that can describe the purpose of the function. The <expr> are an arbitrary number of arbitrary expressions. The optional <return-specifier> gives the return value of the function. It can either be an explicit use of the 'return' function, or it can be any value or expression. Control flow in deffunctions is achieved via the special control-flow expressions 'while' and 'if'. The following is a deffunction that returns the numerically larger of its two arguments:

```
(deffunction max (?a ?b)
  (if (> ?a ?b) then
      (return ?a)
   else
      (return ?b)))
```

3.10 Facts

Jess maintains a list of "facts", or information about the current state of the system. Facts come in two categories: ordered and unordered. Ordered facts are merely lists whose head must be an atom:

```
(temperature 98.6)
(shopping-list bread milk paper-towels)
(start-processing)
```

Unordered facts are more structured; they contain a definite set of 'slots' which must be accessed by name. While ordered facts can be used without prior definition, unordered facts must be defined using the deftemplate construct (see Section 3.11, Deftemplates).

Facts are placed on the fact-list by the 'assert' function. You can see the current fact list using the 'facts' function. You can remove a fact from the fact-list if you know its 'fact ID'. For example,

```
Jess> (assert (foo bar))
  <Fact-0>
Jess> (facts)
  [Fact: foo (ordered) bar]
  For a total of 1 facts.
  TRUE
Jess> (retract 0)
  TRUE
Jess> (facts)
  For a total of 0 facts.
  TRUE
```

3.11 Deftemplates

To define a type of unordered fact, use the deftemplate construct:

```
(deftemplate <deftemplate-name>
  [<doc-comment>]
  (slot <slot-name> [(default <value>)] [(type <typespec>)])
  [(slot ...) ...])
```

The <deftemplate-name> is the head of the facts that will be created using this deftemplate. The <slot-name> must be an atom. The 'default' slot qualifier states that the default value of a slot in a new fact is given by <value>; the default is the atom 'nil'. The 'type' slot qualifier is accepted (for CLIPS compatibility) but ignored by Jess.

As an example, defining the following deftemplate

```
(deftemplate automobile
      "A specific car."
      (slot make)
      (slot model)
      (slot year)
      (slot color (default white)))
```

would allow you to define facts like this:

```
Jess> (assert (automobile (make Chrysler) (model LeBaron) (year 1997)))
  <Fact-1>
Jess> (facts)
  [Fact: automobile (unordered) make=Chrysler; model=LeBaron;
                                year=1997; color=white;]
  For a total of 1 facts.
  TRUE
```

Note that the car is white, by default. Also note that any number of additional automobiles could also be simultaneously asserted onto the fact list using this deftemplate.

A given slot in a deftemplate fact can normally hold only one value. If you want a slot that can hold multiple values, use the 'multislot' keyword instead:

```
(deftemplate box
   (slot location)
   (multislot contents))

(assert (box (location kitchen) (contents spatula sponge frying-pan)))
```

3.12 Deffacts

The deffacts construct is a handy way to define a list of facts that should be made true when the Jess system is started or reset.

```
(deffacts <deffacts-name>
  [<doc-comment>]
  <fact1>
  [...])
```

The deffacts-name is not used; its primary purpose is documentation. A deffacts can contain any number of facts. Any unordered facts in a deffacts must have previously been defined via a deftemplate construct when the deffacts is parsed. The following is a valid deffacts construct:

```
(deffacts automobiles
  (automobile (make Chrysler) (model LeBaron) (year 1997))
  (automobile (make Ford) (model Contour) (year 1996))
  (automobile (make Nash) (model Rambler) (year 1948)))
```

3.13 Defrules

The main purpose of a shell like Jess is to support the execution of rules. Rules in Jess are somewhat like the IF...THEN statements of other programming languages; in operation, Jess constantly tests to see if any of the IFs become true, and executes the corresponding THENs (actually, it doesn't work quite this way, but this is a good way to imagine things. See Section 5, How Jess Works, for a more truthful explanation.) The 'intelligence' embedded in an intelligent rule-based system is encoded in the rules. The defrule construct is used to define a rule to Jess.

```
(defrule <defrule-name>
  [<doc-comment>]
  [<salience-declaration>]
  [[<pattern-binding> <- ] <pattern1>]
  [ (more patterns) ]
  =>
  [<action1> [ <action2> ... ]])
```

Basically, a rule consists of a list of patterns (the IF part) and a list of actions (the THEN part.) The patterns are matched against the fact list. When facts are found that match all the patterns of a rule, the rule becomes activated, meaning it may be fired (have its actions executed) as soon as any other activated rules have been fired. An activated rule may become deactivated before firing if the facts that matched its patterns are retracted, or removed from the fact list, while it is waiting to be fired. Here is an example of a simple rule:

```
(defrule example-1
  "Announce 'a b c' facts"
  (a b c)
 =>
  (printout t "Saw 'a b c'!" crlf))
```

To see this rule in action, enter it at the Jess> prompt, assert the fact (a b c), then the (run) command to start the Jess engine. You'll get some interesting additional information by first issuing the (watch all) command:

```
Jess> (clear)
  TRUE
Jess> (watch all)
  TRUE
Jess>   (defrule example-1
          "Announce 'a b c' facts"
          (a b c)
         =>
          (printout t "Saw 'a b c'!" crlf))
  example-1: +1+1+1+1+t
  TRUE
Jess> (assert (a b c))
  ==>  f-0 [Fact: a (ordered) b c]
  ==> Activation: example-1 : f-0
  <Fact-0>
Jess>  (run)
  FIRE [Defrule: example-1 "Announce 'a b c' facts";
                 1 patterns; salience: 0] f-0
  Saw 'a b c'!
  TRUE
Jess>
```

When you enter the rule, you see the sequence of symbols +1+1+1+1+t. This tells you something about the way that Jess compiled the rule you wrote into the internal rule representation. Then when you assert the fact, Jess responds by telling you that the new fact was assigned the numeric fact identifier 0 (f-0), and that it is an ordered fact with head 'a' and additional fields 'b' and 'c'. Then it tells you that the rule example-1 is activated by the fact f-0, that fact you just entered. When you type (run), you see an indication that your rule has been fired, including a list of the relevant fact IDs. The line "Saw 'a b c'!" is the result the execution of your rule.

If all the patterns of a rule had to be given literally as above, Jess would not be very powerful. Patterns can, however, also include wildcards and various kinds of predicates (comparisons and boolean functions). Firstly, you can specify a variable name instead of a value for a field in any of a rule's patterns (but not the pattern's head.) A variable matches any value in that position within a rule. For example, the rule

```
(defrule example-2
  (a ?x ?y)
  =>
  (printout t "Saw 'a " ?x " " ?y "'" crlf))
```

will be activated each time any fact with head 'a' having two fields is asserted: (a b c), (a 1 2), (a a a), etc. As in the example, the variables thus matched in the patterns (or left-hand-side, LHS) of a rule are available in the actions (right-hand-side, RHS) of the same rule.

Each such variable field in a pattern can also include any number of tests to qualify what it will match. Tests follow the variable name and are separated from it and from each other by ampersands. Tests can either be a literal value (in which case the variable matches only that value,) another variable (which must have been matched earlier in the rule LHS), one of the previous two options preceded by a tilde (~), in which case the test is for inequality, or a colon (:) followed by a function call, in which case the test succeeds if the function returns the special value TRUE (actually in Java it must return a Value object which compares equal to that returned by the static function jess.Funcall.TRUE()). This means you can use any internal or user-defined boolean function as a test. Popular ones are things like eq and neq (comparison) and integerp and stringp (type testing). You can use nested function calls as well; i.e., do arithmetic, then compare the result to a fixed value using eq. Here's an example of a rule that uses several kinds of tests.

```
(defrule example-3
  (not-b-and-c ?n1&~b ?n2&~c)
  (different ?d1 ?d2&~d1)
  (same ?s ?s)
  (more-than-one-hundred ?m&:(> ?m 100))
  =>
  (printout t "Found what I wanted!" crlf))
```

The first pattern will match a fact with head 'not-b-and-c' with exactly two fields such that the first is not 'b' and the second is not 'c'. The second pattern will match any fact with head 'different' and two fields such that the two fields have different values. The third pattern will match a fact with head 'same' and two fields with identical values. The last pattern matches a fact with head 'more-than-one-hundred' and a single field with a numeric value greater than 100.

A few more details about patterns: you can match a field without binding it to a variable by omitting the variable name and using just a question mark (?) as a placeholder. You can match any number of fields using a multivariable (one starting with $?):

```
Jess> (defrule example-4
  (grocery-list $?list)
  =>
  (printout t "I need to buy " $?list crlf))
  TRUE
Jess> (assert (grocery-list eggs milk bacon))
  TRUE
Jess> (run)
  I need to buy (eggs milk bacon)
  TRUE
```

3.13.1 Pattern bindings.

Sometimes you need a handle to an actual fact that helped to activate a rule; for example, when the rule fires, you may need to retract or modify the fact. To do this, you use a pattern-binding variable:

```
(defrule example-5
  ?fact <- (command "retract me")
  =>
  (retract ?fact))
```

The variable (?fact, in this case) is assigned the fact ID of the particular fact that activated the rule.

3.13.2 Salience.

Rules normally file in an unpredictable order, related to but not necessarily the same as the order in which they were activated. To influence this order, rules can include a salience declaration:

```
(defrule example-6
  (declare (salience -100))
  (command exit-when-idle)
  =>
  (printout t "exiting..." crlf))
```

(This rule contains no patterns). Declaring a low salience value for a rule makes it fire after all other rules of higher salience. A high value makes a rule fire before all rules of lower salience. The default salience value is zero.

3.13.3 'Not' patterns.

A pattern can be enclosed in a list with 'not' as the head. In this case, the pattern is considered to match if a fact which matches the pattern is not found. For example:

```
(defrule example-7
   (person ?x)
   (not (married ?x))
   =>
   (printout t ?x " is not married!" crlf))
```

Note that a 'not' pattern cannot contain any variables that are not bound before that pattern (since a 'not' pattern does not match any facts, it cannot be used to define the values of any variables!) A 'not' pattern can similarly not have a pattern binding.

3.14 Defglobals

Jess can support 'global variables' that are visible from the command-prompt or inside any rule or deffunction. You can define them using the defglobal construct:

```
(defglobal
   <varname1> = <value1>
   [<varname2> = <value2> [...]])
```

Note that defglobals, like deffacts, have no effect until a 'reset' command is issued.

4 Things Not Implemented In Jess

Jess does not implement all features of all CLIPS constructs. This list tries to explain some of what's missing from Jess to those who know CLIPS. If you're not already a CLIPS user, you should skip this section!

4.1 Defrules

- Jess implements the simplest form of rule salience. Salience values must be *fixed* integers between -10000 and 10000.
- The 'and' and 'or' conditional elements are not supported on rule LHSs. 'not' is supported, however. You can generally use multiple rules to simulate the effect of an 'and' or 'or' CE.
- The '|' connective constraint is not supported. '&', '~', and predicate constaints (functions like :(eq)) are all supported. Note that instead of writing a pattern like

  ```
  (foo bar|baz)
  ```

  you can write

  ```
  (foo ?X&:(or (eq ?X bar) (eq ?X baz)))
  ```

  to achieve the same effect in Jess.
- The 'test' conditional element is not supported, but it can generally be replaced by a predicate constraint attached to another pattern, i.e,

  ```
  (foo ?X ?Y)
  (test (eq ?X (+ ?Y 3)))
  ```

  can be translated into

  ```
  (foo ?X ?Y&:(eq ?X (+ ?Y 3))).
  ```

- Jess 2.x forced you to name all variables used in patterns. Jess 3.0 now accepts unnamed variables (bare '?' or '$?') for 'don't care' values.

4.2 Deffunctions

- Forward declarations of mutually recursive functions are not needed in Jess and will not parse.

- As of Jess 3.0, Jess, like CLIPS, allows a symbol to be placed at the end of a deffunction, and the value of that symbol will become the return value of the deffunction. The explicit (return) funcall is no longer required.

4.3 Deftemplates

- The only supported slot attribute in Jess is the 'default' attribute. In particular, 'type' will parse, but is ignored at runtime.

4.4 COOL, FuzzyCLIPS, wxCLIPS, etc:

- Jess does not implement any features of these CLIPS extensions.

4 Jess Function Guide

In this Section, every Jess language function shipped with Jess version 3.2 is described. Some of these functions are intrinsic functions, while some are Userfunctions, and may not be available to all Jess code, as detailed above.

*

Arguments:
One or more numeric arguments

Returns:
Number

Description:
Multiplies any number of numeric arguments and returns their product.

**

Arguments:
Two numbers

Returns:
Number

Description:
Raises the first argument to the power of the second using Java's Math.pow() function.

+

Arguments:
One or more numeric arguments

Returns:
Number

Description:
Adds any number of numeric arguments and returns their sum.

-

Arguments:
One or more numeric arguments

Returns:
Number

Description:
Subtracts the second and later arguments from the first, and returns the difference.

/

Arguments:
One or more numeric arguments

Returns:
Number

Description:
Multiplies all but the first argument together, then divides this product into the first argument; returns the quotient.

<

Arguments:
Two or more numeric arguments

Returns:
Boolean

Description:
Returns TRUE if the first argument is less than the second and all later arguments.

<=

Arguments:
Two or more numeric arguments

Returns:
Boolean

Description:
Returns TRUE if the first argument is less than or equal to the second and all later arguments.

<>

Arguments:
Two or more numeric arguments

Returns:
Boolean

Description:
Returns TRUE if the first argument is not equal to any of the second and all later arguments.

=

Arguments:
Two or more numeric arguments

Returns:
Boolean

Description:
Returns TRUE if the first argument is equal to all of the second and later arguments.

>

Arguments:
Two or more numeric arguments

Returns:
Boolean

Description:
Returns TRUE if the first argument is greater than the second and all later arguments.

>=

Arguments:
Two or more numeric arguments

Returns:
Boolean

Description:
Returns TRUE if the first argument is greater than or equal to the second and all later arguments.

abs

Arguments:
One number

Returns:
Number

Description:
Returns the absolute value of the argument.

and

Arguments:
One or more boolean expressions

Returns:
Boolean

Description:
Returns TRUE only if all arguments evaluate to TRUE.

assert

Arguments:
One or more facts (not fact-IDs.)

Returns:
Fact-ID or FALSE

Description:
Asserts all facts onto the fact-list; returns fact-ID of last fact asserted, or FALSE if no facts were successfully asserted (for example, if all facts given are duplicates of existing facts.)

assert-string

Arguments:
One string, containing a representation of a fact.

Returns:
Fact-ID or FALSE

Description:
Attempts to parse string as a fact, and if successful, returns the value returned by assert with the same fact. Note that the string must contain the fact's enclosing parentheses.

batch

Arguments:
One string or atom, the name of a file

Returns:
(Varies)

Description:
Attempts to parse and evaluate the given file as Jess code. If successful, returns the return value of the last expression in the file.

bind

Arguments:
Two, a variable name and any value

Returns:
(Varies)

Description:
Assigns the given value to the given variable, creating the variable if necessary. Note that (as in CLIPS) this works best in rules and deffunctions, and not from the command prompt. Returns the given value.

clear

Arguments:
None

Returns:
TRUE

Description:
Clears Jess. Deletes all rules, deffacts, defglobals, deftemplates, facts, activations, etc. Userfunctions are not deleted.

complement$

Arguments:
Two multifields

Returns:
Multifield

Description:
Returns a multifield consisting of all elements of argument 2 not appearing in argument 1.

create$

Arguments:
Any number of arbitrary values

Returns:
Multifield

Description:
Returns a new multifield containing all the given arguments. Note that multifields must be created explicitly using this function or others that return them; they cannot be directly parsed from Jess input.

delete$

Arguments:
A multifield and two numbers

Returns:
Multifield

Description:
Creates a new multifield by removing elements from the given multifield. The first numeric argument is the one-based index at which to start removing elements; the second is how many elements to remove.

div

Arguments:
Two numbers

Returns:
Numbers

Description:
Quotient of two numbers, properly rounded to the nearest integer.

e

Arguments:
None

Returns:
Number

Description:
Returns the transcendental number 'e'.

eq

Arguments:
Any number of arbitrary arguments

Returns:
Boolean

Description:
Returns TRUE if the first argument is 'equivalent' to all the others. For strings, this means identical contents. Uses the Java Object.equals() function, so can be redefined for external types.

evenp

Arguments:
One integer

Returns:
Boolean

Description:
TRUE if number is an even integer. Results with non-integers may be unpredictable.

exit

Arguments:
None

Returns:
Nothing

Description:
Exits Jess and halts Java.

exp

Arguments:
One number

Returns:
Number

Description:
Returns 'e' raised to the power of the given argument.

facts

Arguments:
None

Returns:
TRUE

Description:
Prints a list of all facts on the fact-list.

first$

Arguments:
One multifield

Returns:
Mutifield

Description:
Returns the first element of the given multifield as a new one-element multifield.

float

Arguments:
One number

Returns:
Floating-point number

Description:
Returns the given argument as a float.

floatp

Arguments:
One number

Returns:
Boolean

Description:
Returns TRUE if the given number has a non-zero fractional component.

gensym*

Arguments:
None

Returns:
Atom

Description:
Returns a unique atom. The atom will consist of the letters "gen" plus an integer. You can set the value of this integer to be used by the next gensym call using setgen (see below.)

halt

Arguments:
None

Returns:
TRUE

Description:
Halts rule execution. No effect unless called from the RHS of a rule.

if

Arguments:
A boolean expression, the atom 'then', and any number of additional expressions; optionally followed by the atom 'else' another list of expression.

Returns:
(Varies)

Description:
The boolean expression is evaluated. If it does not evaluate to FALSE, the first list of expressions is evaluated, and the return value is that returned by the last expression. If it does evaluate to FALSE, and the optional second list of expressions is supplied, those expressions are evaluated and the value of the last is returned.

Example:

```
(if (> ?x 100)
        then
          (printout t "X is big" crlf)
        else
          (printout t "X is small" crlf))
```

implode$

Arguments:
One multifield

Returns:
String

Description:
Converts each element of the multifield to a string, and returns these strings catenated together with single intervening spaces.

insert$

Arguments:
A multifield, an integer, and another multifield

Returns:
A multifield

Description:
Inserts the elements of the second multifield so that they appear starting at the given index of the first multifield.

integer

Arguments:
One number

Returns:
Integer

Description:
Truncates any fractional component of the given number and returns the integral part.

integerp

Arguments:
One number

Returns:
Boolean

Description:
Returns TRUE if the given number has no fractional component.

intersection$

Arguments:
Two multifields

Returns:
Multifield

Description:
Returns a multifield consisting of the elements the two argument multifields have in common.

jess-version-number

Arguments:
None

Returns:
Float

Description:
Returns a version number for Jess; currently 3.2 .

jess-version-string

Arguments:
None

Returns:
String

Description:
Returns a human-readable string descriptive of this version of Jess.

length$

Arguments:
Multifield

Returns:
Integer

Description:
Returns the number of elements in the given multifield.

lexemep

Arguments:
Any value

Returns:
Boolean

Description:
Returns TRUE is the argument is an atom or string.

load-facts

Arguments:
A string or atom, the name of a file of facts

Returns:
Boolean

Description:
The argument should name a file containing a list of facts (not deffacts constructs, and no other commands or constructs.) Jess will parse the file and assert each fact. The return value is the return value of assert when asserting the last fact. In an applet, load-facts will use getDocumentBase() to find the named file.

load-function

Arguments:
One string or atom, the name of a Java class

Returns:
Boolean

Description:
The argument must be the fully-qualified name of a Java class that implements the Userfunction interface. The class is loaded in to Jess and added to the engine, thus making the corresponding command available. See Section 8 on Extending Jess with Java for more information.

load-package

Arguments:
One string or atom, the name of a Java class

Returns:
Boolean

Description:
The argument must be the fully-qualified name of a Java class that implements the Userpackage interface. The class is loaded in to Jess and added to the engine, thus

making the corresponding package of commands available. See Section 8 on Extending Jess with Java for more information.

log

Arguments:
One number

Returns:
Number

Description:
Returns the natural logarithm of the argument.

log10

Arguments:
One number

Returns:
Number

Description:
Returns the base-10 logarithm of the argument.

lowcase

Arguments:
One atom or string.

Returns:
String

Description:
Returns the argument with all characters converted to lower case, as a string.

max

Arguments:
Two numbers

Returns:
Number

Description:
Returns the larger of the two arguments

member$

Arguments:
A value and a multifield

Returns:
Integer or FALSE

Description:
Returns the 1-based index at which the value appears in the multifield, or FALSE if it does not appear.

min

Arguments:
Two numbers

Returns:
Number

Description:
Returns the lesser of the two arguments.

mod

Arguments:
Two integers

Returns:
Integer

Description:
Returns the integral remainder of dividing the first argument by the second.

modify

Arguments:
A fact-ID and any number of two-element lists

Returns:
Fact-ID

Description:
The fact-ID must belong to an unordered fact. Each list is taken as the name of a slot in this fact and a new value to assign to the slot. A new fact is asserted which is similar to the given fact but which has the specified slots replaced with new values. The original fact is retracted. The fact-ID of the new fact is returned.

multifieldp

Arguments:
Any value

Returns:
Boolean

Description:
Returns true if the argument is a multifield.

neq

Arguments:
Two or more values

Returns:
Boolean

Description:
Returns TRUE if the first argument is not equivalent (see eq) to any of the second or remaining arguments.

not

Arguments:
A Boolean expression

Returns:
Boolean

Description:
Returns the Boolean opposite of the argument.

nth$

Arguments:
A number and a multifield

Returns:
(Varies)

Description:
Returns the value at the given 1-based index of the multifield.

numberp

Arguments:
Any value

Returns:
Boolean

Description:
Returns true if the argument is a numeric type.

oddp

Arguments:
One integer

Returns:
Boolean

Description:
Returns TRUE if the argument is an odd number; see evenp.

or

Arguments:
Any number of function calls

Returns:
Boolean

Description:
Returns TRUE if any of the arguments evaluates to TRUE.

pi

Arguments:
None

Returns:
Number

Description:
Returns the number 'pi'.

printout

Arguments:
The atom 't', followed by any number of additional values

Returns:
nil

Description:
Prints its arguments to standard output. The 't' is not printed but must be present. No spaces are added between arguments. The special atom 'crlf' prints as a newline.

random

Arguments:
None

Returns:
Number

Description:
Returns a pseudo-random integer between 0 and 65536.

read

Arguments:
Optionally, the atom 't' (may be omitted).

Returns:
(Varies)

Description:
Reads a single atom, string, or number from standard input, returns this value.

readline

Arguments:
Optionally, the atom 't' (may be omitted).

Returns:
String

Description:
Reads a line from standard input, returns it as a string.

replace$

Arguments:
A multifield, two numbers, and another multifield

Returns:
Multifield

Description:
The second multifield is inserted into the first multifield, replacing elements between the 1-based indices given by the two numeric arguments, inclusive.

Example:

```
Jess> (replace$ (create$ a b c) 2 2 (create$ x y z))
    (a x y z c)
```

reset

Arguments:
None

Returns:
TRUE

Description:
Removes all facts from the fact-list, removes all activations, then asserts the fact (initial-fact), then asserts all facts found in deffacts and initializes all defglobals.

rest$

Arguments:
One multifield

Returns:
Multifield

Description:
Returns a new multifield consisting of all elements from the given multifield except the first.

retract

Arguments:
Any number of fact-IDs

Returns:
TRUE

Description:
Retracts the facts whose IDs are given.

return

Arguments:
One arbitrary value

Returns:
(Varies)

Description:
Returns the given value from a deffunction. Exits the deffunction immediately.

round

Arguments:
One number

Returns:
Integer

Description:
Properly rounds the given number and returns the nearest integer.

rules

Arguments:
None

Returns:
TRUE

Description:
Prints a list of all defrules.

run

Arguments:
None

Returns:
TRUE

Description:
Starts the inference engine. Jess will keep running until no more activations remain or 'halt' is called.

save-facts

Arguments:
A filename, and optionally an atom

Returns:
Boolean

Description:
Attempts to open the named file for writing, and then writes a list of all facts on the fact-list to the file. This file is suitable for reading with load-facts. If the optional second argument is given, only facts whose head matches this atom will be saved. Does not work in applets.

setgen

Arguments:
A number

Returns:
TRUE

Description:
Sets the integer which will be used by gensym* to generate the next unique symbol. Note that if this number has already been used, gensym* uses the next larger number that has not been used.

sqrt

Arguments:
A number

Returns:
Number

Description:
Returns the square root of the argument.

str-cat

Arguments:
Any number of values

Returns:
String

Description:
Converts all arguments to strings and concatenates them together, returning the result as a string.

str-compare

Arguments:
Two strings

Returns:
Integer

Description:
Returns 0 if the strings are identical, -1 if the first is lexically less than the second, +1 if lexically greater.

str-index

Arguments:
Two strings

Returns:
Integer or FALSE

Description:
Returns the 1-based index at which the first string first appears in the second; or FALSE if it does not appear.

str-length

Arguments:
A string

Returns:
Integer

Description:
Returns the length of the string in characters.

stringp

Arguments:
Any value

Returns:
Boolean

Description:
Returns TRUE if the argument is a string.

sub-string

Arguments:
Two numbers and a string

Returns:
String

Description:
Returns the string consisting of the characters between the two 1-based indices of the given string (inclusive).

subseq$

Arguments:
A multifield and two numbers

Returns:
Multifield

Description:
Returns a multifield consisting of the elements between the two 1-based indices of the given multifield (inclusive).

subsetp

Arguments:
Two multifields

Returns:
Boolean

Description:
Returns TRUE if all the elements of the first multifield appear in the second multifield.

sym-cat

Arguments:
Any number of values

Returns:
Atom

Description:
Converts all arguments to strings and concatenates them together, returning the result as an atom.

symbolp

Arguments:
Any value

Returns:
Boolean

Description:
Returns TRUE if the argument is an atom.

system

Arguments:
Any number of values

Returns:
TRUE

Description:
Executes the operating-system command-line constructed by converting each argument to a string.

time

Arguments:
None

Returns:
Number

Description:
Returns the number of seconds since 12:00 AM, Jan 1, 1970.

undefrule

Arguments:
An atom (the name of a rule)

Returns:
Boolean

Description:
Remove the named rule from the Rete network. This rule will never fire again. Returns TRUE if the rule existed.

union$

Arguments:
Two multifields

Returns:
Multifield

Description:
Returns a new multifield consisting of all the elements that appear in the two arguments; duplicates are removed.

unwatch

Arguments:
One of the atoms all, rules, compilations, activations, facts

Returns:
TRUE

Description:
Causes trace output to not be printed for the given indicator. See watch

upcase

Arguments:
A string or atom

Returns:
A string

Description:
Returns the argument as an all-uppercase string.

view

Arguments:
None

Returns:
TRUE

Description:
This Userfunction is included in the Jess distribution but is not normally installed; you must load it using load-function (the class name is jess.view.View). When invoked, it displays a live snapshot of the Rete network in a graphical window. The display is described in Section 5, How Jess Works.

watch

Arguments:
One of the atoms all, rules, compilations, activations, facts

Returns:

TRUE

Description:

Produces additional debug output when specific events happen in Jess, depending on the argument. Any number of different watches can be active simultaneously.

- rules: prints a message when any rule fires.
- compilations: prints a message when any rule is compiled.
- activations: prints a message when any rule is activated, or deactivated, showing which facts have caused the event.
- facts: print a message whenever a fact is asserted or retracted.
- all: all of the above.

while

Arguments:

A function call returning Boolean, the atom 'do', and an arbitrary number of additional function calls.

Returns:

(Varies)

Description:

Evaluates the boolean expression repeatedly. As long as it does not equal FALSE, the list of other expressions are evaluated. The last expression evaluated is the return value.

5 How Jess Works

Note: the information in this Section is provided for the curious reader. An understanding of the Rete algorithm may be helpful in planning expert systems; an understanding of Jess' implementation probably will not be. Feel free to skip this Section and come back to it some other time. You should not take advantage of many of the Java classes mentioned in this Section; they are internal implementation details, and any Java code you write which uses them may well break each time a new version of Jess is released.

Jess is a rule-based expert system shell. In the simplest terms, this means that Jess's purpose it to continuously apply a set of if-then statements, called rules, to a set of data, called the fact list. You define the rules that make up your own particular expert system. Rules in Jess look something like this:

```
(defrule library-rule-1
  (book (name ?X) (status late) (borrower ?Y))
  (borrower (name ?Y) (address ?Z))
 =>
  (send-late-notice ?X ?Y ?Z))
```

Note that this syntax is borrowed from (and is identical to) the syntax used by CLIPS. This rule might be translated into psueudo-english like this:

```
Library rule #1:
If
  a late book exists, with name X, borrowed by someone named Y
and
  that borrower's address is known to be Z
then
  send a late notice to Y at Z about the book X.
```

The book and borrower entities would be found on the fact list. The fact list is therefore a kind of database of bits of factual knowledge about the world. The attributes (called "slots") that things like books and borrowers are allowed to have are defined in statements called "deftemplates"; actions like send-late-notice can be defined in user-written functions in the Jess language ("deffunctions") or in Java ("Userfunctions.") For more information about the CLIPS rule syntax (and to work with Jess, you will certainly need to learn more!) refer to the previous Section, and possibly to the CLIPS documentation as mentioned above.

In a typical expert system a fixed set of rules is used, but the fact list changes continuously. However, it is an empirical fact that in most expert systems, much of the fact list is also fairly fixed; although new facts are arriving and old ones being removed at all times, the percentage of facts that change per unit time is generally fairly small. For this reason, the obvious implementation for the expert system shell is a very inefficient one. This obvious implementation would be to keep a list of the rules, and continuously cycle through the list, checking each one's left-hand-side (LHS) against the fact list, and executing the right-hand-side (RHS) of any rules that apply. This is inefficient because most of the tests made on each cycle will have the same results as on the previous iteration; since the fact list is stable, most of the tests will be repeated. You might call this the 'rules finding facts' approach, and the computational complexity is of the order of O(RPF), where R is the number of rules, P is the average number of patterns per rule LHS, and F is the number of facts on the fact list. This is effectively n^2 in the size of the system.

Jess instead uses a very efficient method known as the Rete (Greek for "net") algorithm. The classic paper on the Rete algorithm *("Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem", Charles L. Forgy, Artificial Intelligence 19(1982), 17-37)* became the basis for a whole generation of fast expert system shells: OPS5, its ancestor ART, and CLIPS. In the Rete algorithm, the inefficiency described above is alleviated (conceptually) by remembering past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs. Additionally, as will be described below, new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity per iteration drops to something more like O(sqrt(RP)). Our discussion of the Rete algorithm is necessarily brief; the interested reader is referred to the Forgy paper or to *Giarrantano and Riley, "Expert Systems: Principles and Programming", Second Edition, PWS Publishing (Boston, 1993)* for a more detailed treatment.

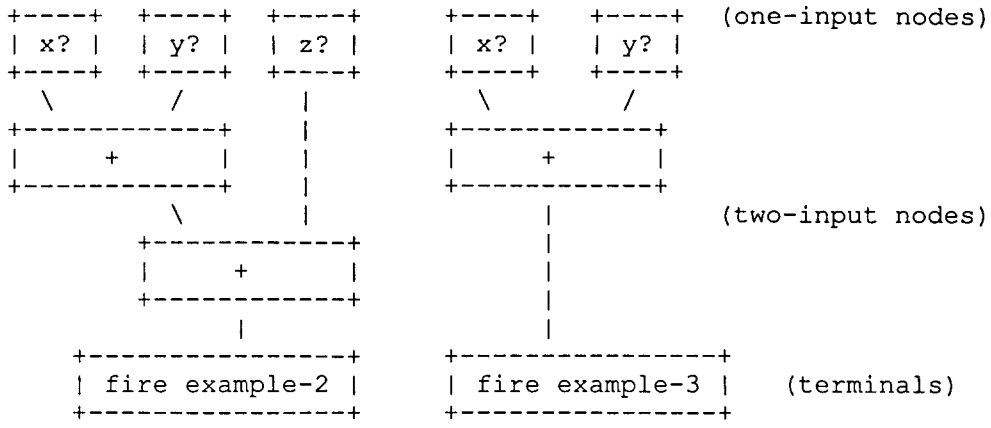
The Rete algorithm is implemented by building a network of nodes, each of which represents one or more tests found on a rule LHS. Facts that are being added to or removed from the fact list are processed by this network of nodes. At the bottom of the network are nodes representing individual rules; when a set of facts filters all the way down to the bottom of the network, it has passed all the tests on the LHS of a particular rule and this set becomes an "activation"; the associated rule may have its RHS executed ("be fired") if the activation is not invalidated first by the removal of one or more facts from its activation set.

Within the network itself there are broadly two kinds of nodes: one-input and two-input nodes. One-input nodes perform tests on individual facts, while two-input nodes perform tests across facts and perform the grouping function. Subtypes of these two classes of node are also used, and there are also auxilliary types such as the terminal nodes mentioned above.

An example is often useful at this point. The following rules:

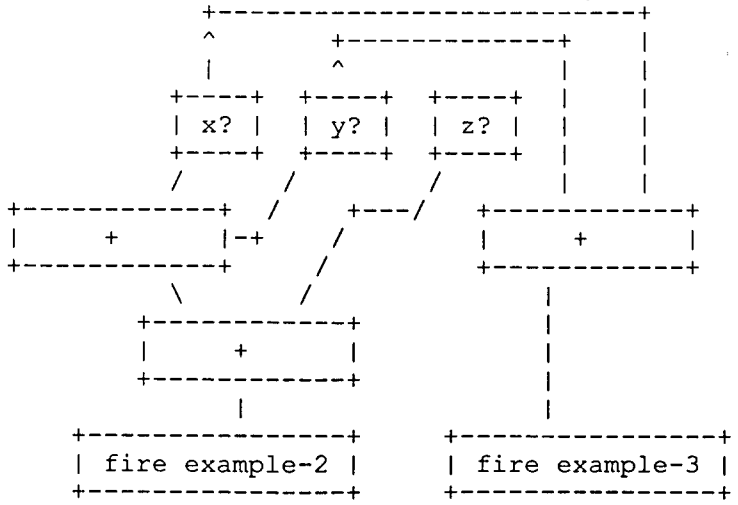
```
(defrule example-2          (defrule example-3
   (x)                         (x)
   (y)                         (y)
   (z)                      => )
 => )
```

might be compiled into the following network:

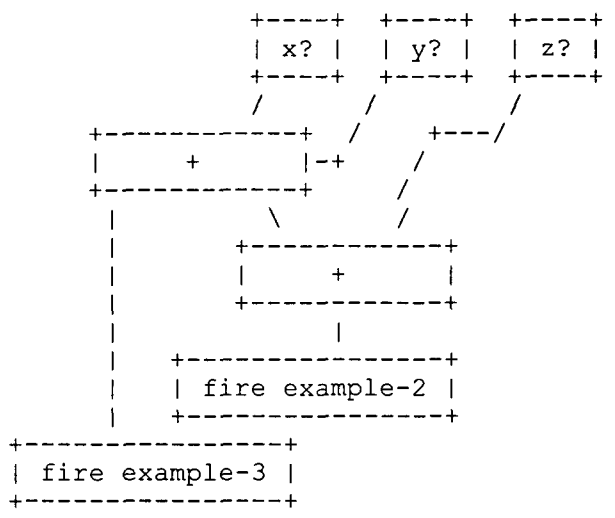


The nodes marked *x?*, etc., test if a fact contains the given data, while the nodes marked + remember all facts and fire whenever they've received data from both their left and right inputs. To run the network, Jess presents new facts to each node at the top of the network as they added to the fact list. Each node takes input from the top and sends its output downwards. A single input node generally receives a fact from above, applies a test to it, and, if the test passes, sends the fact downward to the next node. If the test fails, the one-input nodes simply do nothing. The two-input nodes have to integrate facts from their left and right inputs, and in support of this, their behavior must be more complex. First, note that any facts that reach the top of a two-input node could potentially contribute to an activation: they pass all tests that can be applied to single facts. The two input nodes therefore must remember all facts that are presented to them, and attempt to group facts arriving on their left inputs with facts arriving on their right inputs to make up complete activation sets. A two-input node therefore has a 'left memory' and a 'right memory'. It is here in these memories that the inefficiency described above is avoided. A convenient distinction is to divide the network into two logical components: the single-input nodes comprise the "pattern network", while the two-input nodes make up the "join network".

There are two simple optimizations that can make Rete even better, The first is to share nodes in the pattern network. In the network above, there are five nodes across the top, although only three are distinct. We can modify the network to share these nodes across the two rules (the arrows coming out of the top of the x? and y? nodes are outputs):



But that's not all the redundancy in the original network. Now we see that there is one join node that is performing exactly the same function (integrating x,y pairs) in both rules, and we can share that also:



The pattern and join networks are collectively only half the size they were originally; this kind of sharing comes up very frequently in real systems, and is a significant performance booster!

You can see the amount of sharing in a Jess network by using the 'watch compilations' command. When a rule is compiled and this command has been previously executed, Jess prints a string of characters something like this, which is the actual output from compiling rule example-2, above:

```
example-2: +1+1+1+1+1+1+2+2+t
```

Each time '+1' appears in this string, a new one-input node is created; +2 indicates a new two-input node. Now watch what happens when we compile example-3:

```
example-3: =1=1=1=1=2+t
```

Here we see that =1 is printed whenever a preexisting one-input node is shared; =2 is printed when a two-input node is shared. +t represents the terminal nodes being created. (Note that the number of single-input nodes is larger than expected; Jess creates separate nodes that test for the head of each pattern and its length, rather than doing both of these tests in one node, as we implicitly do in our graphical example.) No new nodes are created for rule example-3; Jess shares existing nodes very efficiently in this case.

Jess's Rete implementation is very literal. Different types of network nodes are represented by various subclasses of the Java class jess.Node: Node1, Node2, NodeNot2, and NodeTerm. The Node1 class is further specialized because it contains a 'command' member which causes it to act differently depending on the tests or functions it needs to perform. For example, there are specializations of Node1 which test the first field (called the 'head') of a fact, test the number of fields of a fact, test single slots within a fact, and compare two slots within a fact. There are further variations which participate in the handling of multifields and multislots. The Jess language code is parsed by the class jess.Jesp, while the actual network is assembled by code in the class jess.ReteCompiler. The execution of the network is handled by the class Rete. The Jess class itself is really just a small demonstration driver for the jess package, in which all of the interesting work is done.

The 'view' command, distributed for the first time with Jess 3.2, is a graphical viewer for the Rete network itself; I have used this as a debugging tool for Jess, but it may have educational value for others, and it may help you to design more efficient systems of rules in

Jess. Issuing the 'view' command after entering the rules example-2 and example-3 produces a very good facsimile of the drawing (although it correctly shows the larger number of one-input nodes.) The various nodes are color-coded according to their roles in the network; Node1 nodes are red; Node2 nodes are green; NodeNot2 nodes are yellow; and NodeTerm nodes are blue (unless they've been 'deactivated' via the undefrule command or being redefined, in which case they are invisible.)Passing the mouse over a node displays information about the node and the tests it contains; double-clicking on a node brings up a dialog box containing the same information. See the description of the view function for important information before using it.

6 Using Jess From Java Code

Using Jess from Java code is simple. The Rete class contains the expert system engine. The Jesp class contains the Jess parser. To execute a file of CLIPS code in Jess (like the CLIPS 'batch' command), simply create a Rete object and a Jesp object, tell the Jesp object about the file, and call Jesp.parse(boolean prompt):

```
// See info about the Display classes in Section 7
NullDisplay nd = new NullDisplay();

// Create a Jess engine
Rete rete = new Rete(nd);

// Open the file test.clp
FileInputStream fis = new FileInputStream("test.clp");

// Create a parser for the file, telling it where to take input
// from and which engine to send the results to
Jesp j = new Jesp(fis, rete);
try
  {
    // parse and execute the code, without printing a prompt
    j.parse(false);
  }
catch (ReteException re)
  {
    // All Jess errors are reported as 'ReteException's.
    re.printStackTrace(nd.stderr());
  }
```

Note that if the file 'test.clp' contains the CLIPS (reset) and (run) commands, the Jess engine will run to completion during the parse() call. Also note that all the classes in the Jess package will throw ReteException exceptions to signal errors.

For more control over Jess from your Java program, you can use the Rete.ExecuteCommand(String) method. For example, after the above code, you could include the following:

```
try
  {
    rete.ExecuteCommand("(reset)");
    rete.ExecuteCommand("(assert (foo bar foo))");
    rete.ExecuteCommand("(run)");
  }
catch (ReteException ex)
  {
    System.err.println("Foo bar error.");
  }
```

I made an effort to make Jess 'sort-of' threadsafe. Sort-of is not nearly good enough, however, so be careful how you use Jess in a multithreaded application. One major difference

between Jess and CLIPS is that you can call (run) from a rule RHS and have a new rule fired up in the middle of RHS execution! This should be used very carefully, if at all.

Jess provides an interface 'ReteDisplay' that provides hooks into the engine's internal workings. The Display provides two types of functions: functions that return an input, output, and error stream to the engine, and functions that are called by the engine whenever an event occurs (events here meaning a construct is parsed, fact is asserted, rule is activated, etc.) You can implement ReteDisplay as a simple way of providing a GUI for your Jess application. The fancy multicolored display for the Jess home page is implemented in the class LostDisplay. Writing a ReteDisplay is the simplest way to customize Jess. See Section 7 for more info about ReteDisplay.

See the file `Jess.java` for ideas on how to implement Jess applets and applications.

7 Capturing and Processing Jess Output

One simple way to use CLIPS as part of a larger system is to capture and process what CLIPS prints on its standard output. This is also possible in Jess. This Section explains how to use the ReteDisplay class to do this.

This is actually very easy to do, but maybe hard to describe. Jess stdin and stdout are Java streams. Jess gets these streams by calling the methods ReteDisplay.stdin() and stdout(). To capture Jess's output, then, all you need to do is to subclass ReteDisplay so that it returns a custom output stream that captures and processes the information in some way. There is an example of this in the Jess distribution, as described below.

Observe that the Jess 'monkey' applet's output appears in a scrolling window. How is this done? The scrolling window is created using an instance of the class TextAreaOutputStream (part of Jess.) TextAreaOutputStream is a subclass of OutputStream that implements all the OutputStream methods such that the data appears in a TextArea Component.

In the distributed 'monkey' applet, Jess.init() creates a TextArea widget, then a TextAreaOutputStream using that widget, then a PrintStream from the TextAreaOutputStream. It then uses this PrintStream to construct a LostDisplay object (a subclass of ReteDisplay) and then uses that LostDisplay to construct a Rete engine. As a result, the output from the Rete engine ends up in the TextArea. (The name LostDisplay comes from the fact that the graphics remind me of the credits from the old TV show 'Lost in Space'.)

So, to capture and process the printed output from Jess, you need to:

1. *Implement an OutputStream class which does what you want to do to the stream of Jess output text.* Look at TextAreaOutputStream to see how. If you want to also print the output as well as process it, you can do that in this class (see the class `java.io.FilterOutputStream` for an elegant way of chaining streams together.)
2. *Implement a ReteDisplay class.* NullDisplay is a very, very simple one which just hands out System.out as stdout(), System.in as stdin(), etc. To create your custom version, you can copy NullDisplay and just add a constructor which lets you pass in your custom streams.
3. In your mainline code, construct an instance of your ReteDisplay using an instance of your OutputStream (remember to coerce it to a PrintStream first). Then use this ReteDisplay to construct your Rete engine. When you run Jess, the printed output will be captured by your stream class.

This all happens in the init() method in the file `Jess.java`.

8 Extending Jess With Commands Written in Java

Jess's rule language can be extended with additional commands written in Java. This, of course, requires you to know the Java programming language, which is not something I can teach you in the confines of this small document. For many real applications, however, extending Jess in this way will be necessary. The good news is that it's very easy, and you can add capabilities to Jess limited only by your imagination.

The class jess.Userfunction represents a single user-supplied function, while the class jess.Userpackage represents a whole set of such functions. Given that you have written some classes which inherit from these classes, you can load these extensions into Jess in two ways. First, you can load them in from Java code. Given that 'rete' is the Rete object in your application, and 'myfunction' is the name of a Userfunction class you (or someone else!) wrote, you can add the new function to Jess by calling

```
rete.AddUserfunction(new myfunction());
```

or an entire package of such functions in a class 'mypackage' using

```
rete.AddUserpackage(new mypackage());
```

Starting in Jess 3.1, you can load functions and packages form the Jess language itself. The equivalents to the above are

```
(load-function "myfunction")
```

and

```
(load-package "mypackage")
```

Note that if the new classes or user packages come in a Java package, you'll need to specify the fully qualified name of the class:

```
(load-package "xyzzy.bassomatic.mypackage")
```

In any case, the relevant classes need to be reachable on your Java CLASSPATH.

I've made it as easy as possible to add user-defined functions to Jess. There is no system type-checking on arguments, so you don't need to tell the system about your arguments, and values are self-describing, so you don't need to tell the system what type you return. You do, however, need to understand two basic Jess data structures: class `Value` and class `ValueVector`. Look at the source for these classes if the following discussion isn't clear.

8.1 The class `jess.Value`

A `Value` is a self-describing data object. Once it is constructed, its type and value cannot be changed. `Valuesupports` a `type()` function, which returns one of these type constants (defined in the class `jess.RU`, 'Rete Utilities'):

```
final public static int NONE             =     0;
final public static int ATOM             =     1;
final public static int STRING           =     2;
final public static int INTEGER          =     4;
final public static int VARIABLE         =     8;
final public static int FACT_ID          =    16;
final public static int FLOAT            =    32;
final public static int FUNCALL          =    64;
final public static int ORDERED_FACT     =   128;
final public static int UNORDERED_FACT   =   256;
final public static int LIST             =   512;
final public static int DESCRIPTOR       =  1024;
final public static int EXTERNAL_ADDRESS =  2048;
final public static int INTARRAY         =  4096;
final public static int MULTIVARIABLE    =  8192;
final public static int SLOT             = 16384;
final public static int MULTISLOT        = 32768;
```

`Value` objects are constructed by specifying the data and the type. Each overloaded constructor assures that the given data and the given type are compatible. Note that for each constructor, more than one value of the `type` parameter is acceptable. The available constructors are:

```
public Value(Object o, int type) throws ReteException
public Value(String s, int type) throws ReteException
public Value(Value v)
public Value(ValueVector f, int type) throws ReteException
public Value(double d, int type) throws ReteException
public Value(int value, int type) throws ReteException
public Value(int[] a, int type) throws ReteException
```

`Value` supports a number of functions to get the actual data out of a `Value`object. These are

```
public Object ExternalAddressValue() throws ReteException
public String StringValue() throws ReteException
public ValueVector FactValue() throws ReteException
public ValueVector FuncallValue() throws ReteException
public ValueVector ListValue() throws ReteException
public double FloatValue() throws ReteException
public double NumericValue() throws ReteException
public int AtomValue() throws ReteException
public int DescriptorValue() throws ReteException
public int FactIDValue() throws ReteException
public int IntValue() throws ReteException
public int VariableValue() throws ReteException
public int[] IntArrayValue() throws ReteException
```

If you try to convert random values by creating a Value and retrieving it as some other type, you'll generally get a ReteException. However, many types can be freely interconverted: Strings and atoms, for example, or integers and floats.

Note that Jess stores all strings, atoms and variable names as integers, which are used as indexes into a hashtable. Thus if `Value.type()` returns RU.ATOM or RU.STRING, you can call either `AtomValue()` (which returns that integer) or `StringValue()` (which returns a Java String object.) To convert a String to an appropriate integer, call `int RU.putAtom(String)`. To get the String that goes with an integer, call `String RU.getAtom(int)`. Note that this is NOT a way to convert the String "1" to the integer 1; it converts Strings into unique hash codes.

8.2 The class `jess.ValueVector`

Facts, function calls, lists, etc. are stored by Jess in objects of class `jess.ValueVector`. `ValueVector` is an extensible array of `Value` objects. You set an element of a `ValueVector` with `void set(Value, int)` and get an element with `Value get(int)`. `set()` and `get()` will throw an exception if the index you're accessing is past the end of the current array. You can add a value to the end of a `ValueVector` with `void add(Value)` (which can extend the length of the internal data structures.) `int size()` returns the actual number of `Values` in the `ValueVector`. `void set_length(int)` lets you cheat by extending the length of a `ValueVector` to include null `Values`. (This is necessary sometimes to allow filling in many elements in random order.)

Facts (type RU.ORDERED_FACT or RU.UNORDERED_FACT) are stored as a ValueVector with the slots filled in a special way, as follows (the constants representing slot numbers MUST be used, as they may change)

SLOT NUMBER	TYPE	DESCRIPTION
RU.CLASS	RU.ATOM	The 'head' or first field of

		the fact
RU.ID	RU.FACT_ID	The fact-id of this fact
RU.DESC	RU.DESCRIPTOR	One of RU.ORDERED_FACT or RU.UNORDERED_FACT
RU.FIRST_SLOT	(ANY)	Value of the first slot of this fact
RU.FIRST_SLOT + 1	(ANY)	second
...	...	...

Note that for ordered facts, the slots are stored in the order in which they appear, but in unordered (deftemplate) facts, they appear in the order given in the corresponding deftemplate.

Function calls (RU.FUNCALLs) are simpler; the first slot is the functor as an RU.ATOM, and all remaining slots are arguments. When your user function is called, the first argument to the Java Call function will be a ValueVector representation of the Jess code that evoked your function.

Now, on to writing a user function. First, create a class that implements the interface jess.Userfunction, which just contains the two methods `name()` and `Call()`. A listing is worth a 1000 words:

```
// A user function that implements the CLIPS 'upcase' operation in
// terms of the java.lang.String.toUpperCase() method.

class MyUpcase implements jess.Userfunction
  {

    private int _name = RU.putAtom("upcase");

    // The name method returns the integer representation of the name.
    // This function will be called by Jess.
    public int name() { return _name; }

    public Value Call(ValueVector vv, Context context) throws ReteException
      {
        return new Value(vv.get(1).StringValue().toUpperCase(), RU.STRING);
      }
  }
```

Note that we use `vv.get(1).StringValue()` to get the first argument to 'upcase' as a java String. If the argument doesn't contain a string, a ReteException will be thrown that describes the problem; hence you don't need to worry about incorrect argument types. `vv.get(0)` will always return 'upcase', the name of the function being called. `vv.get(1)` is the first argument, `vv.get(2)` would be the second, if this function accepted multiple arguments. If you want, you can check how many arguments your function was called with and throw a ReteException if it was the wrong number.

Then in your mainline code, simply call Rete.AddUserfunction() with an instance of your new class as an argument, and the function will be available from Jess code. Adding to our mainline code from the last section:

```
// Add the 'upcase' command to Jess
rete.AddUserfunction(new upcase());
// Exceute some Jess code that calls this function
rete.ExecuteCommand("(printout t (upcase foo) crlf)");
```

will print "FOO".

Jess 3.0 added the `jess.Userpackage` interface. `jess.Userpackage` is a handy way to group a collection of `Userfunctions` together. A `Userpackage` class should supply the one

method `Add()`, which adds a collection of `Userfunctions` to a Rete engine using `AddUserfunction()`. Nothing mysterious going on, but it's very convenient. Implementations for strcat, strcompare, etc, are found in the sample file `jess/StringFunctions.java`.

```
public class StringFunctions implements Userpackage {

  public void Add(Rete engine) {
    engine.AddUserfunction(new strcat());
    engine.AddUserfunction(new upcase());
    engine.AddUserfunction(new lowcase());
    engine.AddUserfunction(new strcompare());
    engine.AddUserfunction(new strlength());
    engine.AddUserfunction(new substring());

  }
}
```

Now in your mainline, you can call

```
engine.AddUserpackage(new StringFunctions());
```

and from your Jess code, you can call str-cat, str-compare, etc.

There are a lot of new small classes in the Jess package which serve as examples of Userfunctions. These days, with zips and JAR files, this isn't such a big deal. Still, you can leave them out if you want just by removing the line that adds the relevant Userpackage from your mainline program.

9 The Future of Jess

Jess will continue to be maintained and improved for the foreseeable future. I have a list of features I plan to implement, but it's hard to associate timescales with any of them. They are listed in order. The first few are likely to appear in the next few months as a Jess 3.3 release. The later ones... who knows? I don't expect a Jess 4.0 release before the end of this year. For Jess 3.3:

- The (test) conditional element
- Parsing of integers (right now all parsed numbers are floats)

For Jess 4.0?

- A much more extensive Java API for embedding Jess in other applications
- Direct access to Java methods and variables from CLIPS code (using the Java Reflection API)
- Some subset of COOL functionality, possibly in the form of pattern matching on Java member fields
- Optional compilation of jess rules to pure Java code (potential for large speed improvements)

10 Version History

Version 3.2

system and integer Userfunction classes renamed (Win95 filename capitalization problem!) Broken delete$, insert$, replace$ fixed. 'view' command added. Big if/then in Funcall class finally removed in favor of separate implementation classes for intrinsics, leading to a modest speed increase. Documentation vastly expanded! Added catch for ArrayOutOfBoundsException in command-line interface; no more crash on wrong number of args. Broken evenp, oddp fixed. str-cat, sym-cat made more general. Broken sub-string fixed. Big switch in Node1 class replaced by separate classes, leading to a very modest speed increase.

Version 3.1:

Added the 'assert-string' and 'batch' commands. Two bug fixes in multislot code (thanks to Nancy Flaherty). Added 'undefrule' and the ability to redefine rules. Added the 'system' function, although it doesn't work very well under Java. Public function engine() in jess.Context class allows you to do fancier things in UserFunctions. Added the non-standard 'load-package' and 'load-function' functions. Many new contributed functions packaged with Jess for doing math, handling multifields, and other neat stuff; thanks to Win Carus for these. Added 'time' (1 second resolution).

Version 3.0:

A few code changes to accomodate Microsoft's Java compiler; Jess now compiles unchanged with JVC (thanks to Mike Finnegan.) Added 'member$' multifield function. Added 'clear' intrinsic (thanks to Karl Mueller.) Introduced a new way of handling (not) patterns which I think finally guarantees there are no more not-related bugs remaining! 'load-facts', which has been non-functional throughout the beta period, is working again. Documentation now explains unzipping and compiling a little better. Modified the way fact-id's are handled so that you can write '(retract 3)' to retract fact #3.

Version 3.0b2:

LOTS of bug reports and improvement suggestions from the field - thanks folks! All the reported bugs in the multifield implementation, and some residual odd behavior in the "not" CE, have been fixed. The (exit) command has been added. A command prompt has been added. The '#' character can now be used in symbols. The access levels on some methods in the Rete class have been opened up; Rete is no longer final. nth$ is now 1-based, as it is in CLIPS. The "if" and "while" constructs now fire on 'not FALSE' instead of 'TRUE'. The str-index function has been fixed and added. Probably a few more things I'm forgetting here. Thanks for the input; particular thanks to Nancy Flaherty, Joszef Toth, Karl Mueller, Duane Steward, and Michelle Dunn for reporting bugs fixed in this version; sorry if I left anyone out.

Version 3.0b1:

First public release of Jess 3.0.

Version 3.0a3:

UserPackage interface; lots of new example UserFunctions for multifields, string, and predicates.

Version 3.0a2:

Multislots! Also important bug fix: under certain circumstances, the Rete network compilation could fail 1) if (not ()) CEs occurred on the LHS of a rule, 2) new variables were introduced in that rule's patterns listed after the (not ()) CEs, and 3) these latter variables were tested (i.e., in a predicate constraint) on the LHS of the rule.

Version 3.0a1:

Incremental reset. Watch activations. gc() in LostDisplay, NullDisplay. Multifields! All the Rete engine classes are now in a package named 'jess'. Many classes and methods that should not be manipulated by clients are now package-private.

Version 2.2.1:

Ken Bertapelle found another bug, which has been squashed, in the pattern network.

Version 2.2:

Jess 2.2 adds a few new function calls (load-facts, save-facts) and fixes a serious bug (thanks to Ken Bertapelle for pointing it out!) which caused Jess to crash when predicate constraints were used in a certain way. Another bugfix corrected the fact that 'retract' only retracted the first of a list of facts. Jess used to give a truly

inscrutable error message if a variable was first used in a not CE (a syntax error); the current error message is much easier to understand. I also clarified a few points in the documentation.

Version 2.1:

Jess 2.1 is *much* faster than version 2.0. The Monkey example runs in about half the time as under Jess 2.0, and for some inputs, the speed has increased by an order of magnitude! This is probably the last big speed increase I'll get. For Java/Rete weenies, this speed increase came from banishing the use of java.lang.Vector in Tokens and in two-input node memories. Jess is now within a believeable interpreted Java/C++ speed ratio range of about 30:1. Jess 2.1 now includes rule salience. It also implements a few additional intrinsic functions: gensym*, mod, readline. Jess 2.1 fixes a bug in the way predicate constraints were parsed under some conditions by Jess 2.0. The parser now reports line numbers when it encounters an error.

Version 2.0:

Jess 2.0 is intrinsically about 30% faster than version 1.0. The internal data structures changed quite a bit. The Rete network now shares nodes in the Join network instead of just in the pattern network. The current data structures should allow for continued improvement.

If anyone writes an emulation of a CLIPS function that Jess omits, *please* send it to me and I'll include it in the next release (with credit to you, of course.)

At the time of this writing, Jess has more than 5000 registered users. I have been very pleased by this response and have enjoyed working with many of Jess's more ambitious users. If you use Jess, and if you have comments, questions, or concerns, please don't hesitate to ask.

Finally, thanks to Gary Riley and the gang at NASA for writing the marvelous CLIPS in the first place!

DISTRIBUTION

1	MS 9001	Thomas Hunter, 8000 Attn: J.B. Wright, 2200 J.F. Ney (A), 5200 W.J. McLean, 8300 R.C. Wayne, 8400 P.N. Smith, 8500 P.E. Brewer, 8600 T.M. Dyer, 8700 L.A. Hiles, 8800
1	MS 9003	D.L. Crawford, 8900
1	MS 9004	M.E. John, 8100
1	MS 9214	L.M. Napolitano, 8130
1	MS 9012	J.E. Costa, 8920
20	MS 9214	E.J. Friedman-Hill, 8920
1	MS 9012	R.A. Whiteside, 8920
1	MS 0807	Rich Detry, 4418
1	MS 9012	Carmen Pancerella, 8920
1	MS 9420	Robert Mariano, 8220
1	MS 9420	Paul Klevgard, 8220
1	MS 9420	Scot Marburger, 8220
1	MS 9420	Jim Smith, 8220
1	MS 9420	Barry Hess, 8220
1	MS 9214	Nina Berry, 8920
1	MS 0722	John Mitchiner, 6534
1	MS 0188	Donna Chavez, LDRD Office
3	MS 9018	Central Technical Files, 8940-2
4	MS 0899	Technical Library, 4916
1	MS 9021	Technical Communications Department, 8815/Technical Library, MS 0899, 4916
2	MS 9021	Technical Communications Department, 8815 for DOE/OSTI